We discuss the Bentley–Ottmann sweep line algorithm for line segment intersection and give an implementation of the function

```
SWEEP_SEGMENTS(const list&lt;SEGMENT&gt;& S, GRAPH&lt;POINT,SEGMENT&gt;& G,
               bool embed, bool use_optimization)
```

that takes a list S of segments and computes the graph G induced by it. For each vertex v of G it also computes its position in the plane, and for each edge e of G it computes the segment containing it.

If *embed* $=$ *true*, the algorithm turns G into a planar map, i.e., G is made bidirected and the adjacency lists are sorted according to the geometric embedding in clockwise order.

If *use_optimization* $=$ *true*, an optimization described below is used.

The algorithm runs in time $O((n+s)\log(n+m)+m)$, where n is the number of segments, s is the number of nodes of G, and m is the number of edges of G. If S contains no overlapping segments then $m = O(n+s)$. If *embed* is *true*, the running time is increased by an additive factor of $O(m \log m)$. Note that $s \leq 3(n+k)$ and that k can be as large as s^2.

We want to stress that the implementation makes no assumptions about the input, in particular, segments may have length zero, may be vertical or may overlap, several segments may intersect in the same point, endpoints of segments may lie in the interior of other segments, … .

We achieve this generality by reformulating the plane sweep algorithm so that it can handle all geometric situations. The reformulation makes the description of the algorithm shorter and it also makes the algorithm faster, since k is replaced by s in the time bound[1]. The only previous algorithm that could handle all degeneracies is due to Myers [Mye85]. Its expected running time for random segments is $O(n \log n + k)$ and its worst case running time is $O((n+k)\log n)$.

In the sweep line paradigm a vertical line is moved from left to right across the plane and the output (here the graph $G(S)$) is constructed incrementally as it evolves behind the sweep line. One maintains two data structures to keep the construction going: the so-called *Y-structure* contains the intersection of the sweep line with the scene (here the set S of line segments) and the so-called *X-structure* contains the events where the sweep has to be stopped in order to add to the output or to update the X- or Y-structure. In the line segment intersection problem an event occurs when the sweep line hits an endpoint of some segment or an intersection point. When an event occurs, some nodes and edges are added to the graph $G(S)$, the Y-structure is updated, and maybe some more events are generated. When the input is in general position (no three lines intersecting in a common point, no endpoint lying on a segment, no two endpoints or intersections having the same x-coordinate, no vertical lines, no overlapping segments, …) then at most one event can occur for each position of the sweep line and there are three clearly distinguishable types of events (left endpoint, right endpoint, intersection) with easily describable associated actions, cf. [Meh84, VII.8]. We

[1] Bentley and Ottmann formulated their algorithm for line segments in general position and stated a time bound of $O((n+k)\log n)$.

want to place no restrictions on the input and therefore need to proceed slightly differently. We now describe the required changes.

We define the sweep line by a point *p_sweep* $= (x\_sweep, y\_sweep)$. Let ϵ be a positive infinitesimal (readers not familiar with infinitesimals may think of ϵ as an arbitrarily small positive real). Consider the directed line L consisting of a vertical upward ray ending in point $(x\_sweep + \epsilon^2, y\_sweep + \epsilon)$ followed by a horizontal segment ending in $(x\_sweep - \epsilon^2, y\_sweep + \epsilon)$ followed by a vertical upward ray. We call L the *sweep line*. Note that[2] no endpoint of any segment lies on L, that no two segments of S intersect L in the same point except if the segments overlap, and that no non-vertical segment of S intersects the horizontal part of L. All three properties follow from the fact that ϵ is arbitrarily small but positive. Figure 0.1 illustrates the definition of L and the main data structures used in the algorithm: the Y-structure, the X-structure, and the graph G.

The Y-structure contains all segments intersecting the sweep line L ordered as their intersections with L appear on the directed line L. Overlapping segments are ordered by their *ID-numbers*. Every segment has an associated ID-number; distinct segments are guaranteed to have distinct IDs.

The X-structure contains all endpoints that are to the right of the sweep line and also some intersection points between segments in the Y-structure. More precisely, for each pair of segments adjacent in the Y-structure their intersection point is contained in the X-structure (if it exists and is to the right of the sweep line). The X-structure may contain other intersection points. The graph G contains the part of $G(S)$ that is to the left of the sweep line.

Initially, the Y-structure and the graph G are empty and the X-structure contains all endpoints of all input segments. The events in the X-structure are then processed in left to right order. Events with the same x-coordinate are processed in bottom to top order.

Assume that we need to process an event at point p and that the X-structure and Y-structure reflect the situation for *p_sweep* $= (p.x, p.y - 2\epsilon)$. Note that this is true initially, i.e., before the first event is removed from the X-structure. We now show how to establish the invariants for *p_sweep* $= p$. We proceed in seven steps.

1. We add a node v at position p to our graph G.
2. We determine all segments in the Y-structure containing the point p. These segments form a possibly empty subsequence of the Y-structure.
3. For each segment in the subsequence we add an edge to the graph G.
4. We delete all segments ending in p from the Y-structure.
5. We update the order of the subsequence in the Y-structure. This amounts to moving the sweep line across the point p.
6. We insert all segments starting in p into the Y-structure.
7. We generate events for the segments in the Y-structure that become adjacent by the actions above and insert them into the X-structure.

[2] We defined the sweep line in this seemingly complicated way in order to be able to write this “Note that”. The note will allow us to define a linear order on the segments intersecting the sweep line.

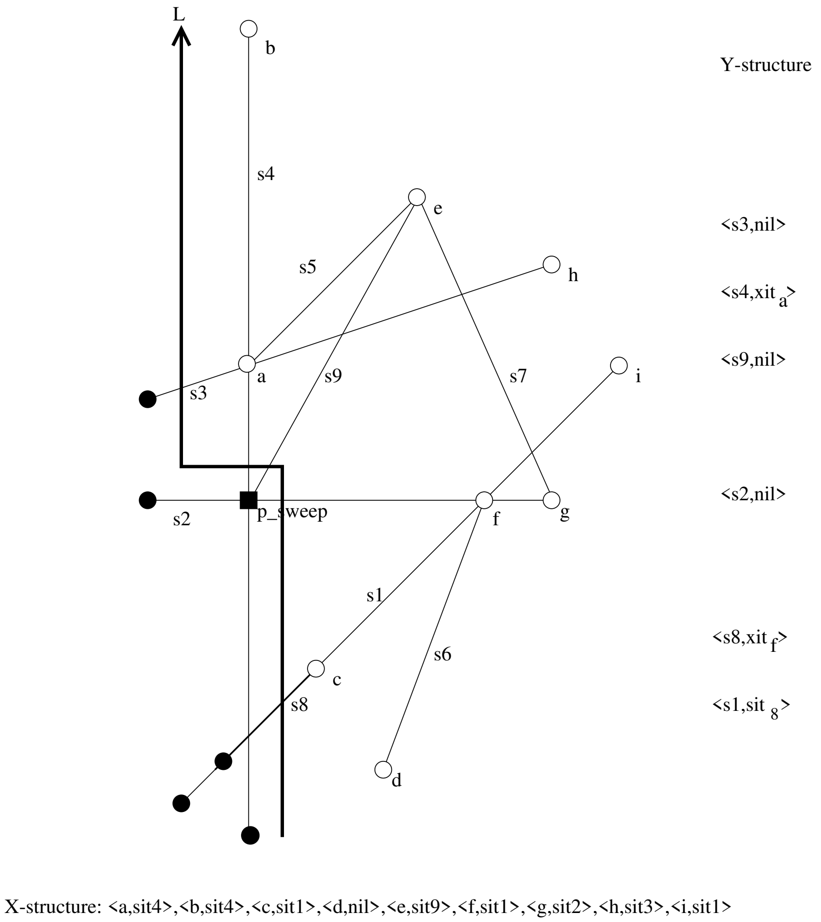

Figure 0.1 A scene of nine segments. The segments s_1 and s_8 overlap. The sweep line is shown in bold. The part of $G(S)$ to the left of the sweep line is already constructed. Its nodes are shown filled. The sweep line intersects the segments s_1, s_8, s_2, s_9, s_4, and s_3 and in this order. The Y-structure contains one item for each one of them. The X-structure contains points a, b, c, d, e, f, g, h, and i and in this order.
The information associated with the items in the X- and Y-structure will be explained in the next section.

This completes the description of how to process the event p. The invariants now hold for *p_sweep* $= p$ and hence also for *p_sweep* $= (p'.x, p'.y - 2\epsilon)$ where p' is the new first element of the X-structure.

0.0.1 *The Implementation of the Sweep Line Algorithm*

This section is joint work with Ulrike Bartuschka.

The implementation follows the algorithm closely. It makes use of several data types discussed in earlier chapters. The main “ingredients” are the basic geometric objects and primitives, sorted sequences for the X- and Y-structure, priority queues for storing events, and graphs for representing the output.

To make this section self-contained we briefly review the data types used.

Points and Segments: The types *rat_point* and *rat_segment* realize points and segments in the plane with rational coordinates and are part of the rational kernel. A *rat_point* is specified by its homogeneous coordinates of type *integer* – the type of arbitrary precision integers. If p is a *rat_point* then $p.X(\,)$, $p.Y(\,)$, and $p.W(\,)$ return its homogeneous coordinates and *p.xcoord*( ) and *p.ycoord*( ) return its Cartesian coordinates. If x, y, and w are of type *integer* with $w \neq 0$ then *rat_point*(x, y) and *rat_point*(x, y, w) create the *rat_point* with homogeneous coordinates $(x, y, 1)$ and (x, y, w), respectively. Two points are equal (*operator*==) if they agree in their Cartesian coordinates. A *rat_segment* is specified by its two endpoints; so if p and q are *rat_points* then *rat_segment*(p, q) is the directed segment with source p and target q. If s is a *rat_segment* then *s.source*( ) and *s.target*( ) return the source and target of s, respectively.

There are also points (class *point*) with coordinates of type *double*. The corresponding segment class is called *segment*. The classes *point* and *segment* have the same interface as *rat_point* and *rat_segment*. However, the internal representation is different: instead of storing the homogeneous coordinates as *integers*, the Cartesian coordinates are stored as *doubles*.

The sweep program can be executed with either the rational or the floating point geometry kernel. Be aware, however, that the instantiation with the floating point kernel is not fully reliable, see Section . In the sequel we use POINT to denote the point class and SEGMENT to denote the segment class.

POINTS and SEGMENTS come with a large number of geometric primitives. In the sweep program the following primitives are used:

- *int compare*(*POINT* p, *POINT* q)
  compares points by their lexicographic order; p precedes q if either
  p.xcoord( ) < *q.xcoord*( ) or
  p.xcoord( ) = *q.xcoord*( ) and *p.ycoord*( ) < *q.ycoord*( ). The function returns -1 if p precedes q, returns 0 if p and q are equal, and returns $+1$ otherwise. The lexicographic order of points is the default order on points.

- *int orientation*(*POINT* p, *POINT* q, *POINT* r)
  computes the orientation of points p, q, and r in the plane, i.e., 0 if the points are collinear, -1 if they define a clockwise oriented triangle, and $+1$ if they define a counter-clockwise oriented triangle.

- *int orientation*(*SEGMENT* s, *POINT* p)
  computes *orientation*(*s.source*( ), *s.target*( ), p).

- *int cmp_slopes*(*SEGMENT s1*, *SEGMENT s2*) compares the slopes of *s1* and *s2*. If one of the segments is degenerate, i.e., has length zero, the result is zero. Otherwise, the result is the sign of $slope(s1) - slope(s2)$.

- *bool intersection_of_lines*(*SEGMENT s1*, *SEGMENT s2*, *POINT*& *p*)
  returns *false* if segments *s1* and *s2* are parallel or one of them is degenerate. Otherwise, it computes the point of intersection of the two straight lines supporting the segments, assigns it to the third parameter q, and returns *true*.

Our program maintains its own set of segments which we call *internal segments* or simply segments and store in the list *internal*; input segments are called input segments or original segments when the need for distinction arises. Internal segments are directed from left to right; vertical segments are directed upwards. There is one internal segment for every non-trivial input segment. The *map*<*SEGMENT*, *SEGMENT*> *original* stores for each internal segment the corresponding original segment.

⟨*local declarations*⟩≡

`list<SEGMENT>         internal;`

`map<SEGMENT,SEGMENT> original;`

Sorted Sequences: The type *sortseq*<K, I> realizes sorted sequences of pairs in $K \times I$, see Section **??**; K is called the key type and I is called the information type of the sequence. The key type must be linearly ordered, i.e., the function *int compare*(*const K*&, *const K*&) must be defined for the type K and the relation $<$ on K defined by $k_1 < k_2$ iff $compare(k_1, k_2) < 0$ must be a linear order on K. An object of type *sortseq*<K, I> is a sequence of items (type *seq_item*) each containing a pair in $K \times I$. We use <k, i> to denote an item containing the pair (k, i) and call k the key and i the information of the item. The keys in a sorted sequence $\langle k_1, i_1\rangle, \langle k_2, i_2\rangle, \ldots, \langle k_m, i_m\rangle$ form an increasing sequence, i.e., $k_l < k_{l+1}$ for $1 \le l < m$.

Let S be a sorted sequence of type *sortseq*<K, I> and let k and i be of type K and I, respectively. The operation *S.lookup*(k) returns the item $it = \langle k, .\rangle$ in S with key k if there is such an item and returns *nil* otherwise. If *S.lookup*(k) == *nil* then *S.insert*(k, i) adds a new item $\langle k, i\rangle$ to S and returns this item. If *S.lookup*(k) == *it* then *S.insert*(k, i) changes the information in the item *it* to i. If $it = \langle k, i\rangle$ is an item of S then *S.key*(*it*) and *S.inf*(*it*) return k and i, respectively, and *S.succ*(*it*) and *S.pred*(*it*) return the successor and predecessor item of *it*, respectively; the latter operations return *nil* if these items do not exist. The operation *S.min*( ) returns the first item of S, *S.empty*( ) returns *true* if S is empty and *false* otherwise. Finally, if *it1* and *it2* are items of S with *it1* before *it2* then *S.reverse_items*(*it1*, *it2*) reverses the subsequence of S starting at item *it1* and ending at item *it2*.

In our implementation the X-structure has type *sortseq*<*POINT*, *seq_item*> and the Y-structure has type *sortseq*<*SEGMENT*, *seq_item*>. The Y-structure has one item for each segment intersecting the sweep line. The information field in the Y-structure is used for cross-links with the X-structure and for linking overlapping segments.

The X-structure is ordered according to the default order of points and the Y-structure is ordered according to the intersections of the segments with the directed sweep line L. The position of the sweep line is determined by *p_sweep* and the comparison object *cmp* realizes the order in the Y-structure. The class *sweep_cmp* will be defined below.

⟨*local declarations*⟩+≡

```
POINT p_sweep;
sweep_cmp cmp(p_sweep);
sortseq<POINT,seq_item>     X_structure;
sortseq&lt;SEGMENT, seq_item&gt;  Y_structure(cmp);
```

In the example of Figure 0.1 the sweep line intersects the segments s_1, s_8, s_2, s_9, s_4, and s_3. The Y-structure therefore consists of six items, one each for segments s_1, s_8, s_2, s_9, s_4, and s_3.

The X-structure contains an item for each endpoint of an input segment that is to the right of the sweep line and an item for each intersection point between segments that are adjacent in the Y-structure and that intersect to the right of the sweep line. It may also contain intersection points between segments that are not adjacent in the Y-structure.[3] The points in the X-structure are ordered according to the lexicographic ordering of their Cartesian coordinates. As mentioned above this is the default order on points.

In the example of Figure 0.1 the X-structure contains items for the endpoints b, c, d, e, g, h, i and for intersections a and f. Here, a and f are the intersections between segments s_4 and s_3, and s_1 and s_2, respectively.

The informations associated with the items of both structures serve as cross-links between the two structures: the information associated with an item in the X-structure is either *nil* or an item in the Y-structure; the information associated with an item in the Y-structure is either *nil* or an item of either structure. The precise definition follows: consider first an item $\langle s, it\rangle$ in the Y-structure and let s' be the segment associated with the successor item it' in the Y-structure. If s and s' overlap then $it = it'$. If s and s' do not overlap and $s \cap s'$ exists and lies to the right of the sweep line then it is the item in the X-structure with key $s \cap s'$. In all other cases we have $it = nil$.

Consider next an item $\langle p, sit\rangle$ in the X-structure. If $sit \neq nil$ then sit is an item in the Y-structure and the segment associated with it contains p. Moreover, if there is a pair of adjacent segments in the Y-structure that intersect in p then $sit \neq nil$. We may have $sit \neq nil$ even if there is no pair of adjacent segments intersecting in p.

In our example, the Y-structure contains the items $\langle s_1, sit_8\rangle$, $\langle s_8, xit_f\rangle$, $\langle s_2, nil\rangle$, $\langle s_9, nil\rangle$, $\langle s_4, xit_a\rangle$, and $\langle s_3, nil\rangle$ where sit_8 is the item of the Y-structure with associated segment s_8 and xit_a and xit_f are the items of the X-structure with associated points a and f, respectively. Let's turn to the items of the X-structure next. All items except $\langle d, nil\rangle$ point back to the Y-structure. If sit_i denotes the item $\langle s_i, \ldots\rangle$, $i \in \{1, 2, 9, 4, 3\}$, of the Y-structure

[3] Our X-structure may contain intersection points between segments that are no longer adjacent in the Y-structure. These events could be removed from the X-structure. Removing these events would guarantee an X-structure of linear size, however, at the cost of complicating the code. Since the size of the X-structure is always bounded by the size of the output graph we do not remove these events.

then the items of the X-structure are $\langle a, sit_4\rangle$, $\langle b, sit_4\rangle$, $\langle c, sit_1\rangle$, $\langle d, nil\rangle$, $\langle e, sit_9\rangle$, $\langle f, sit_1\rangle$, $\langle g, sit_2\rangle$, $\langle h, sit_3\rangle$, and $\langle i, sit_1\rangle$.

The Order on the Y-structure: The segments in the Y-structure are ordered according to their intersection with the sweep line. Overlapping segments are ordered according to their ID-number. All segments in the Y-structure are non-trivial and the position of the sweep line is determined by *p_sweep*.

The Y-structure is realized as a sorted sequence. In a sorted sequence comparisons between keys are only made during insertions and lookups and then one of the keys involved in the comparison is an argument of the operation. We conclude that compare is only called for segments s_1 and s_2 where one of the segments has its source point equal to *p_sweep*. Also, at least one of the segments is non-trivial and if one of the segments is trivial it has both endpoints equal to *p_sweep*. Let us assume first that both segments are non-trivial.

Assume s_i has its source point equal to *p_sweep*. If *p_sweep* does not lie on s_{1-i}, i.e., $orientation(s_{1-i}, p\_sweep) \neq 0$, then the orientation test is also the outcome of compare.

If both segments contain *p_sweep* we compare the slopes of s_1 and s_2 (*orientation*(*s2*, *s1.target*( ))). Only overlapping segments are equal after this comparison. They are ordered according to their ID-numbers. Since only internal segments are stored in the Y-structure and since internal segments are pairwise non-identical, any two internal segments have different ID-numbers.

The compare class *sweep_cmp* is derived from *leda_cmp_base*, see Section **??**. It has a private data member *p_sweep* whose value will always be equal to the position of the sweep line; in the constructor the data member is initialized to the initial position of the sweep line and *set_position* is used to inform the compare object about any advance of the sweep line.

⟨*geometric primitives*⟩≡

```
class sweep_cmp : public leda_cmp_base&lt;SEGMENT&gt;
{
  POINT p_sweep;
public:
 sweep_cmp(const POINT& p) : p_sweep(p) {}
 void set_position(const POINT& p) { p_sweep = p; }
 int operator()(const SEGMENT& s1, const SEGMENT& s2) const
 { // Precondition:
   // p_sweep is identical to the left endpoint of either s1 or s2.
   if (identical(s1,s2)) return 0;
   int s = 0;
   if ( identical(p_sweep,s1.source()) )  s = orientation(s2,p_sweep);
   else
    if ( identical(p_sweep,s2.source()) ) s = orientation(s1,p_sweep);
    else error_handler(1,"compare error in sweep");
   if (s || s1.is_trivial() || s2.is_trivial()) return s;
   s = orientation(s2,s1.target());
   // overlapping segments will be ordered by their ID_numbers :
```

```
    return s ? s : (ID_Number(s1) - ID_Number(s2));
  }
 };
```

We still need to explain the purpose of the tests *is_trivial*. We will also have to locate trivial segments in the Y-structure. These segments will have both endpoints equal to *p_sweep*. We want the search to be successful iff the Y-structure contains a segment passing through *p_sweep*. In the order defined above, the trivial segment (*p_sweep*, *p_sweep*) is larger than all segments intersecting the sweep line before *p_sweep*, is equal to all segments passing through *p_sweep*, and is larger than all segments intersecting the sweep line after *p_sweep*. We conclude that a search for the trivial segment will return a segment passing through *p_sweep* if there is one.

It is important to observe that the compare function for segments changes as the sweep progresses. What does it mean then that the keys of the items in a sorted sequence form an increasing sequence? The requirement is that whenever a lookup or insert operation is applied to a sorted sequence, the sequence must be sorted with respect to the current compare function. The other operations may be applied even if the sequence is not sorted.

The Graph G: The graph G has type *GRAPH<POINT, SEGMENT>*, i.e., it is a directed graph where a *POINT*, respectively *SEGMENT*, is associated with each node, respectively edge, of the graph. The graph G is the part of $G(S)$ that is left of the sweep line. The point associated with a vertex defines its position in the plane and the segment associated with an edge is an input segment containing the edge. We use two operations to extend the graph G. If p is a *POINT* then *G.new_node*(p) adds a new node to G, associates p with the node, and returns the new node. If v and w are nodes of G and s is a *SEGMENT* then *G.new_edge*(v, w, s) adds the edge (v, w) to G, associates s with the edge, and returns the new edge. In order to facilitate the addition of edges we maintain a *map<SEGMENT, node> last_node*: it gives for each segment in the Y-structure the rightmost vertex lying on the segment.

⟨*local declarations*⟩+≡

```
map&lt;SEGMENT,node&gt;                 last_node(nil);
```

The Priority Queue: We use a priority queue *seg_queue* to drive the insertion of segments into the Y-structure. The queue contains all internal segments that are ahead of the sweep line ordered according to their left endpoint. In particular, the first segment in *seg_queue* is always the segment that is encountered next by the sweep line. *Seg_queue* has type *p_queue<POINT, SEGMENT>*.

The data type *p_queue*$<P, I>$ realizes priority queues with priority type P and information type I. P must be linearly ordered. Priority queues are an item-based data type. Every item (of type *pq_item*) stores a pair (p, i) from $P \times I$, p is called the priority and i is called the information of the item. The usual operations on priority queues (*insert*, *delete_min*, *find_min*) are available.

⟨*local declarations*⟩+≡
```
p_queue<POINT,SEGMENT>      seg_queue;
```

We are now ready for the program. It has the following structure:

⟨*sweep_segments.c*⟩+≡
⟨*geometric primitives*⟩
⟨*embedding*⟩
```
void SWEEP_SEGMENTS(const list&lt;SEGMENT&gt;& S, GRAPH&lt;POINT, SEGMENT&gt;& G,
                    bool embed, bool use_optimization)
```
{ ⟨*local declarations*⟩
⟨*initialization*⟩
⟨*sweep*⟩
⟨*post processing*⟩
}

Initialization: We describe the initialization of the data structures. We clear the graph G, we compute a coordinate *Infinity* that is larger than the absolute value of the coordinates of all endpoints and that plays the role of ∞ in our program, we insert the endpoints of all input segments into the X-structure, and we create for each non-trivial input segment an internal segment with the same endpoints, insert this segment into *seg_queue* and link the input segment to it (through map *original*), we create two sentinel segments at $-\infty$ and $+\infty$, respectively, and insert them into the Y-structure, we put the sweep line at its initial position by setting *p_sweep* to $(-\infty, -\infty)$, and we add a stopper point with coordinates $(+\infty, +\infty)$ to *seg_queue*. The sentinels avoid special cases and thus simplify the code. Finally, we introduce a variable *next_seg* that always contains the first segment in *seg_queue*.

⟨*initialization*⟩≡
```
G.clear();
COORD Infinity = 1;
SEGMENT s;
forall(s,S)
{
  COORD x1 = s.xcoord1(), y1 = s.ycoord1();
  COORD x2 = s.xcoord2(), y2 = s.ycoord2();
  if (x1 < 0) x1 = -x1;
  if (y1 < 0) y1 = -y1;
  if (x2 < 0) x2 = -x2;
  if (y2 < 0) y2 = -y2;
  while (x1 >= Infinity || y1 >= Infinity ||
         x2 >= Infinity || y2 >= Infinity )   Infinity *= 2;
  seq_item it1 = X_structure.insert(s.source(), seq_item(nil));
  seq_item it2 = X_structure.insert(s.target(), seq_item(nil));
  if (it1 == it2) continue;  // ignore zero-length segments
  POINT p = X_structure.key(it1);
  POINT q = X_structure.key(it2);
  SEGMENT s1 = ( compare(p,q) < 0 ? SEGMENT(p,q) : SEGMENT(q,p) );
```

```
    original[s1] = s;
    internal.append(s1);
    seg_queue.insert(s1.source(),s1);
  }
  SEGMENT lower_sentinel(-Infinity,-Infinity,Infinity,-Infinity);
  SEGMENT upper_sentinel(-Infinity, Infinity,Infinity, Infinity);
  p_sweep = lower_sentinel.source();
  cmp.set_position(p_sweep);
  Y_structure.insert(upper_sentinel,seq_item(nil));
  Y_structure.insert(lower_sentinel,seq_item(nil));
  POINT pstop(Infinity,Infinity);
  seg_queue.insert(pstop,SEGMENT(pstop,pstop));
  SEGMENT next_seg = seg_queue.inf(seg_queue.find_min());
```

There is one subtle point in the code above. An insert operation into a sorted sequence with a key that is already present in the sorted sequence returns the item containing the key; it does not add a new item to the sequence and its does not change the key of the item returned. We exploit this feature of sorted sequences to ensure that internal segments share endpoints. Assume for concreteness that s_1 and s_2 are two input segments with a common source point and assume that s_1 is processed first. When the source point of s_2 is inserted into the X-structure, the item containing the source point of s_1 will be returned and hence the internal segments corresponding to s_1 and s_2 have the same (not just equal) source point[4].

Processing Events: We now come to the heart of procedure sweep: processing events. Let $event = \langle p, sit\rangle$ be the first event in the X-structure and assume inductively that our data structure is correct for $p\_sweep = (p.x,\ p.y - 2\epsilon)$. Our goal is to change *p_sweep* to p, i.e., to move the sweep line across point p. As long as the X-structure is not empty we perform the following actions.

We first extract the next event point *p_sweep* from the X-structure by assigning the minimal key in the X-structure to *p_sweep*, adjusting the compare function for segments to the new position of the sweep line, and adding a vertex v with position *p_sweep* to the output graph G. Then, we handle all segments passing through or ending at *p_sweep*. Finally, we insert all segments starting at *p_sweep* into the Y-structure, check for possible intersections between pairs of segments now adjacent in the Y-structure, and update the X-structure. Finally, we delete the event from the X-structure.

⟨*sweep*⟩≡

```
  while ( !X_structure.empty() )
  { seq_item event = X_structure.min();
    p_sweep = X_structure.key(event);
    cmp.set_position(p_sweep);
    node v = G.new_node(p_sweep);
    ⟨handle passing and ending segments⟩
```

[4] A point is realized as a pointer to a representation class. Two points are equal if they have the same Cartesian coordinates and two points are identical if they share the representation. Testing two points for identity is faster than testing them for equality.

```
  ⟨insert starting segments⟩
  ⟨compute new intersections and update X-structure⟩
  X_structure.del_item(event);
}
```

Handling Passing and Ending Segments: We first determine the segments passing through or ending in *p_sweep* and then handle them by reversing their order in the Y-structure.

⟨*handle passing and ending segments*⟩≡

```
seq_item sit = X_structure.inf(event);
if (sit == nil) sit = Y_structure.lookup(SEGMENT(p_sweep,p_sweep));
seq_item sit_succ  =     nil;
seq_item sit_pred  =     nil;
seq_item sit_pred_succ = nil;
seq_item sit_first =     nil;
if (sit != nil)
{ ⟨determine passing and ending segments⟩
  ⟨reverse order of passing segments⟩
}
```

We first determine whether there is any segment passing through or ending in *p_sweep*. Recall that the current event is ⟨*p_sweep*, *sit*⟩.

If $sit \neq nil$, the segment associated with *sit* contains *p_sweep*. If $sit = nil$, there is no pair of adjacent non-overlapping segments in the Y-structure intersecting in *p_sweep*. However, there may be a bundle of overlapping segments in the Y-structure that contain *p_sweep*. We can decide whether there is such a bundle and determine some segment in the bundle by locating the point *p_sweep* in the Y-structure[5]. We defined the comparison function for segments such that a search for the trivial segment (*p_sweep*, *p_sweep*) in the Y-structure is successful iff the Y-structure contains a segment containing *p_sweep*.

If there is no segment in the Y-structure containing *p_sweep*, there is nothing to do. Assume otherwise. Then *sit* points to one such segment. We determine all such segments. The corresponding items form a subsequence of the Y-structure, see Figure 0.2. We compute the first (*sit_first*) and last (*sit_last*) item of this bundle of items and also the predecessor (*sit_pred*) and successor (*sit_succ*) item of the bundle. We also store in *sit_pred_succ* the successor of *sit_pred* before the insertion, i.e, *sit_first*.

The items in the bundle are easily recognized by their informations. The information of every item in the bundle except for the last is either equal to the current event item *event* or equal to the successor item in the Y-structure (in the case of a segment overlapping with its successor). The information of the last item in the bundle is either *nil* or an item in the X-structure different from *event* (such an item stands for an intersection with *sit_succ*).

We determine the items in the bundle as follows. Starting at *sit* we first walk up until

[5] The Y-structure contains segments and hence only segments can be located in it. In order to locate the point *p_sweep* in the Y-structure, we locate the zero-length segment (*p_sweep*, *p_sweep*) instead.

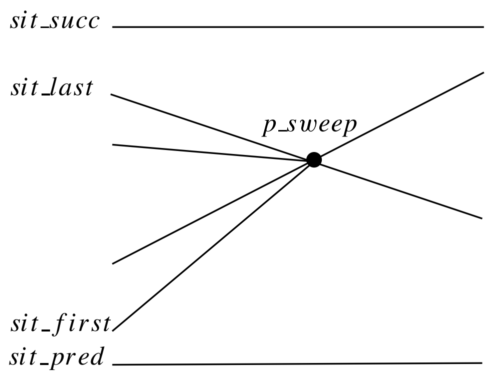

Figure 0.2 The items *sit_pred*, *sit_first*, *sit_last*, and *sit_succ*.

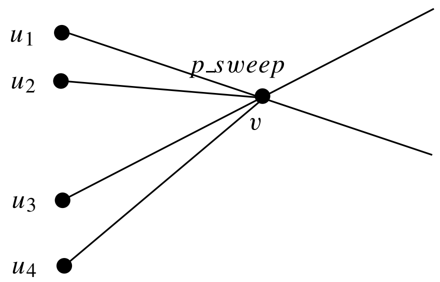

Figure 0.3 The edges out of v are constructed in the order (v, u_1), (v, u_2), (v, u_3), (v, u_4).

sit_succ is reached. Then we walk down to *sit_pred*. During the downward walk we also start to update the data structures. For every segment s in the bundle we do the following:

- We add an edge to G connecting *last_node*[s] and v and label it with s. The new edge gets its direction from the original segment containing it, if *embed* is *false*, and is directed from v to *last_node*[s], if *embed* is *true*.
- If s ends at *p_sweep* then we delete it from the Y-structure. If the predecessor segment overlaps with s, we copy the information about the successor segment of s (if any) to the predecessor and set a flag that the downward walk is not finished yet.
- If s continues through *p_sweep* then we change the intersection information associated with it to *nil* and set *last_node* to v.

We explain why we direct the edge constructed for a segment s from v to *last_node*[s] if *embed* is *true*. Since *new_edge* appends the edge constructed to the list of outgoing edges of v and since we construct edges during the downward walk the edges out of v will be constructed in their proper counter-clockwise order, see Figure 0.3. We will exploit this fact when we construct the planar embedding of G in the post-processing step.

The identification of the subsequence of segments incident to *p_sweep* takes constant time per element of the sequence. Moreover, the constant is small since the test of whether p is incident to a segment involves no geometric computation but only identity tests between items. The code is particularly simple due to our sentinel segments: *sit* can never be the first or last item of the Y-structure.

⟨*determine passing and ending segments*⟩≡

```
// walk up
while ( Y_structure.inf(sit) == event ||
        Y_structure.inf(sit) == Y_structure.succ(sit) )
  sit = Y_structure.succ(sit);
sit_succ = Y_structure.succ(sit);
seq_item sit_last = sit;
if ( use_optimization ) { ⟨optimization, part 1⟩ }
// walk down
bool overlapping;
do
{ overlapping = false;
  s = Y_structure.key(sit);
  if ( !embed && s.source() == original[s].source() )
    G.new_edge(last_node[s], v, s);
  else
    G.new_edge(v, last_node[s], s );
  if ( identical(p_sweep,s.target()) )   // ending segment
  {
     seq_item it = Y_structure.pred(sit);
     if ( Y_structure.inf(it) == sit )
     { overlapping = true;
       Y_structure.change_inf(it, Y_structure.inf(sit));
     }
     Y_structure.del_item(sit);
     sit = it;
   }
   else  // passing segment
   {
     if ( Y_structure.inf(sit) != Y_structure.succ(sit) )
       Y_structure.change_inf(sit, seq_item(nil));
     last_node[s] = v;
     sit = Y_structure.pred(sit);
   }
} while ( Y_structure.inf(sit) == event || overlapping ||
          Y_structure.inf(sit) == Y_structure.succ(sit) );
sit_pred = sit;
sit_first = Y_structure.succ(sit_pred);
sit_pred_succ = sit_first;
```

All segments in the bundle starting with *sit_first* and ending in *sit_last* pass through node v and moving the sweep line through *p_sweep* changes the order of these segments in the Y-structure. More precisely, if s and s' are two segments passing through *p_sweep* then moving the sweep line through *p_sweep* reverses their order iff s and s' do not overlap.

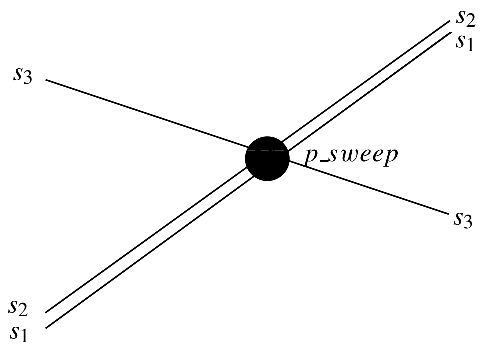

Figure 0.4 Three segments passing through *p_sweep*, two of them overlapping. The order of the segments is reversed, but the order within the sub-bundle of overlapping segments is retained.

If the bundle is non-empty, we update its order as follows: first we reverse all subsequences of overlapping segments and then we reverse the entire bundle, see Figure 0.4.

The bundle of segments passing through *p_sweep* is empty iff *sit_first* is equal to *sit_succ*.

⟨*reverse order of passing segments*⟩≡

```
sit = sit_first;
// reverse subsequences of overlapping segments  (if existing)
while ( sit != sit_succ )
{ seq_item sub_first = sit;
  seq_item sub_last  = sub_first;
  while (Y_structure.inf(sub_last) == Y_structure.succ(sub_last))
    sub_last = Y_structure.succ(sub_last);
  if ( sub_last != sub_first )
    Y_structure.reverse_items(sub_first, sub_last);
  sit = Y_structure.succ(sub_first);
}
// reverse the entire bundle
if ( sit_first != sit_succ )
  Y_structure.reverse_items(Y_structure.succ(sit_pred),
                            Y_structure.pred(sit_succ));
```

Insertion of Starting Segments: The last step in handling the event point *p_sweep* is to insert all segments starting at *p_sweep* into the Y-structure and to test the new pairs of adjacent items (*sit_pred*, . . .) and (. . . , *sit_succ*) for possible intersections. If there were no segments passing through or ending in *p_sweep* then the items *sit_succ* and *sit_pred* still have the value *nil* and we have to compute them now.

We use the priority queue *seg_queue* to find the segments to be inserted. As long as the first segment in *seg_queue* starts at *p_sweep*, i.e., *next_seg.source*( ) is identical[6] to *p_sweep*,

[6] Recall that we ensured that endpoints of internal segments that are equal are identical.

we remove it from the queue and locate it in the Y-structure. Let *s_it* be the item returned by *locate* and let *p_sit* be its predecessor.

We insert *next_seg* after *s_it* into the Y-structure; this will add an item *sit* to the Y-structure. We set the information of *sit* to *s_sit* if the new segment overlaps with the segment associated with *s_sit* and we set it to *nil* otherwise. Similarly, if the new segment overlaps with the segment associated with *p_sit* we change the information of *p_sit* to *sit*.

We associate the new item *sit* with the right endpoint of *next_seg* in the X-structure; note that the point is already there but it does not have its link to the Y-structure yet. We also set *last_node*[s] to v, and if *sit_succ* and *sit_pred* are still undefined, i.e, there was no segment passing through or ending in *p_sweep*, we set them to the successor and predecessor of the new item, respectively, and we set *sit_pred_succ* to *sit_succ*.

⟨*insert starting segments*⟩≡

```
while ( identical(p_sweep,next_seg.source()) )
{ seq_item s_sit = Y_structure.locate(next_seg);
  seq_item p_sit = Y_structure.pred(s_sit);

  s = Y_structure.key(s_sit);

  if ( orientation(s, next_seg.start()) == 0 &&
       orientation(s, next_seg.end()) == 0 )
     sit = Y_structure.insert_at(s_sit, next_seg, s_sit);
  else
     sit = Y_structure.insert_at(s_sit, next_seg, seq_item(nil));

  s = Y_structure.key(p_sit);

  if ( orientation(s, next_seg.start()) == 0 &&
       orientation(s, next_seg.end()) == 0 )
    Y_structure.change_inf(p_sit, sit);

  X_structure.insert(next_seg.end(), sit);
  last_node[next_seg] = v;

  if ( sit_succ == nil )
  { sit_succ = s_sit;
    sit_pred = p_sit;
    sit_pred_succ = sit_succ;
  }

  // delete minimum and assign new minimum to next_seg

  seg_queue.del_min();
  next_seg = seg_queue.inf(seg_queue.find_min());
}
```

Computing New Intersections: If *sit_pred* still has the value *nil*, there were no ending, passing or starting segments and hence *p_sweep* is an isolated point and we are done. Isolated points result from segments of length zero.

So assume that *sit_pred* exists. We have to update its information field (which still has the value from before the event). We set it to *nil* if there is no intersection between *sit_pred* and its successor. If the intersection exists, we insert it into the X-structure and set the information field of *sit_pred* to it. If there are segments leaving *p_sweep*, i.e, *sit_pred* is not the predecessor of *sit_succ*, we also check for an intersection between *sit_succ* and its predecessor.

⟨*compute new intersections and update X-structure*⟩≡

```
if ( sit_pred != nil )
{ if ( !use_optimization )
  { Y_structure.change_inf(sit_pred,seq_item(nil));
    compute_intersection(X_structure, Y_structure, sit_pred);
    sit = Y_structure.pred(sit_succ);
    if ( sit != sit_pred )
      compute_intersection(X_structure, Y_structure, sit);
  }
  else
  { ⟨optimization, part 2⟩ }
}
```

The function *compute_intersection* takes an item *sit0* of the Y-structure and determines whether the segment associated with *sit0* intersects the segment associated with its successor item *sit1* to the right of the sweep line. If so, it updates the X- and the Y-structure. Let s_0 and s_1 be the segments associated with *sit0* and *sit1*, respectively, and let ℓ_0 and ℓ_1 be the supporting lines of s_0 and s_1, respectively.

We know that s_0 intersects the sweep line L before s_1. Thus s_0 and s_1 intersect right of the sweep line if the right endpoint of s_1 lies below or on ℓ_0 (*orientation*(*s0*, *s1.target*( )) ≥ 0) and the right endpoint of s_0 lies above or on ℓ_1 (*orientation*(*s1*, *s0.target*( )) ≤ 0).

If the segments intersect, we compute the point of intersection, call it q, by a call of *s0.intersection_of_lines*(*s1*, q), insert a new pair (q, *sit0*) into the X-structure and associate this pair with *sit0* in the Y-structure.

⟨*geometric primitives*⟩+≡

```
static void compute_intersection(sortseq<POINT,seq_item>& X_structure,
                   sortseq&lt;SEGMENT,seq_item&gt;& Y_structure, seq_item sit0)
{ seq_item sit1 = Y_structure.succ(sit0);
  SEGMENT  s0   = Y_structure.key(sit0);
  SEGMENT  s1   = Y_structure.key(sit1);

  if ( orientation(s0,s1.target()) <= 0 &&
       orientation(s1,s0.target()) >= 0 )
  { POINT q;
    s0.intersection_of_lines(s1,q);
    Y_structure.change_inf(sit0, X_structure.insert(q,sit0));
  }
}
```

Post Processing: We associate with each edge of G an input segment containing it. This is easily done as each edge has an internal segment associated with it. Thus we only have to replace $G[e]$ by *original*[$G[e]$].

The graph G constructed during the sweep is planar but is not in the form of a planar map yet. In particular, the order of the adjacency lists depends on the insertion order.

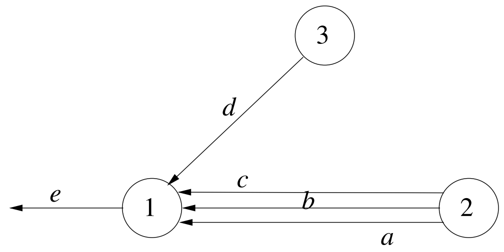

Figure 0.5 Before the call of embedding there is only one edge leaving node 1, namely, the edge e. There are three parallel edges (2, 1); their counter-clockwise order around node 2 is in decreasing order of ID-number. We need to add the reversals of the edges a, b, c, and d to the list of edges out of 1. Sorting the edges by increasing slope and edges of equal slope by ID-number gives the desired order.

When *embed* is *true*, we turn G into a planar map.

⟨*post processing*⟩≡
```
if (embed) construct_embedding(G);
edge e;
forall_edges(e, G) G[e] = original[G[e]];
```

When *embed* is *true* all edges of G are directed from right to left (vertical edges are directed downwards). Moreover, the edges out of any node are already in their proper counter-clockwise order.

In order to turn G into a planar map we need to add the reversal of every edge and to insert the new edges at their proper position into the adjacency lists.

Edge reversals are directed from left to right (the reversal of a vertical edge is directed upwards). The proper order of edge reversals is therefore by slope. Reversals of parallel edges should be ordered by ID-number. Consider Figure 0.5.

Let R be a copy (!!!) of the set of all edges of G. We use R instead of E to indicate that R represents the set of edge reversals. We sort the edges in R according to slope and then add for each edge e in R the edge $(target(e), source(e))$ to G. Since new edges are appended to the lists of outgoing edges, this will result in properly ordered adjacency lists.

⟨*embedding*⟩≡
```
class sweep_cmp_edges : public leda_cmp_base<edge>
{
  const GRAPH<POINT,SEGMENT>& G;
public:
  sweep_cmp_edges(const GRAPH<POINT,SEGMENT>& g): G(g) {}
  int operator()(const edge& e1, const edge& e2) const
  { SEGMENT s1 = G[e1];
    SEGMENT s2 = G[e2];
    int c = cmp_slopes(s1,s2);
    if (c == 0) c = compare(ID_Number(s1),ID_Number(s2));
    return c;
```

```
    }
  };

  static void construct_embedding(GRAPH<POINT,SEGMENT>& G)
  {
    list<edge> R = G.all_edges();
    sweep_cmp_edges cmp(G);
    R.sort(cmp);
    edge e;
    forall(e,R)
    { edge r = G.new_edge(target(e),source(e),G[e]);
      G.set_reversal(e,r);
    }
  }
```

In the post-processing step we first compute the embedding and then replace internal segments by input segments. It would be incorrect to change the order of two steps: first, the ordering of the Y-structure is an ordering on internal segments and we must use the same ordering in the embedding step. Second, the input may contain multiple occurrences of the same segment and the ordering by ID-number does not break ties between identical segments.

An Optimization: The running time of SWEEP_SEGMENTS is $O((n+s)\log(n+m)+m)$ where n is the number of segments, s is the number of nodes of G and m is the number of edges of G. If there are no overlapping segments then $m = O(n+s)$ since G is planar. In the presence of overlapping segments, m may be as large as $n(n+s)$. The time bound can be seen as follows. There are $O(n+k)$ lookups, insertions, and deletions in the X- and Y-structure, each for a cost of $O(\log(n+m))$. Observe that $n+m$ is an upper bound on the number of items in the Y-structure and that $n+s$ is an upper bound on the number of items in the X-structure. Since $s \leq n^2$ we have $\log(n+s+m) = O(\log(n+m))$. The total number of items handled by the *reverse_items* operations on the Y-structure is $O(m)$. Since the cost of *reverse_items* is proportional to the number of items reversed, the total cost for all *reverse_items* operations is $O(m)$. The number of operations on G is $O(n+k+m)$, each for a cost of $O(1)$.

Experiments show that a significant fraction of the running time is spent in the geometric primitives *sweep_cmp* and *compute_intersection*, in particular, if the rational kernel is used (which we recommend). The rational kernel has a built-in floating point filter, i.e., all geometric tests are first performed in floating point arithmetic, the rounding error is estimated, and only if the error estimation indicates that the result of the floating point computation may be wrong, the computation is repeated with exact arithmetic. The floating point filter is discussed in detail in Section **??**.

The function *compute_intersection* performs orientation tests and computes an intersection point. The floating point filter applies to the orientation tests but does not apply to the computation of intersection points since constructions of new points are always performed with exact arithmetic.

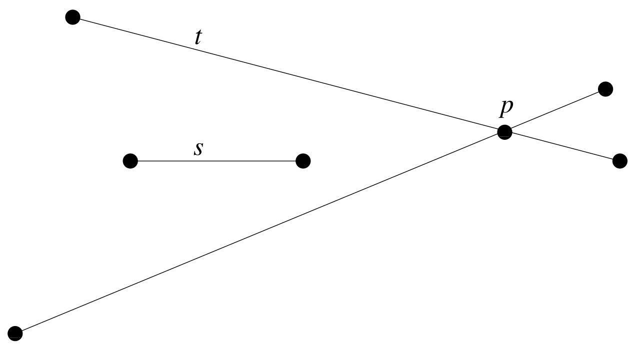

Figure 0.6 The intersection p is first discovered when t is inserted into the Y-structure and is rediscovered when s is removed from the Y-structure.

The function *compute_intersection* is called whenever two segments become adjacent in the Y-structure. Segments may become adjacent in the Y-structure more than once, see Figure 0.6. We show how to avoid the recomputation of intersections.

We maintain a dictionary *inter_dic* which maps pairs of segments to items in the X-structure. The appropriate data type is a two-dimensional map.

⟨*local declarations*⟩+≡

```
map2&lt;SEGMENT,SEGMENT,seq_item&gt;   inter_dic(nil);
```

Whenever a pair of segments that is adjacent in the Y-structure becomes non-adjacent we store their intersection in the dictionary and whenever a pair of segments becomes adjacent we consult the dictionary to find out whether their intersection was already computed.

When processing an event two intersections may get lost. Consider the sequence of items corresponding to segments passing through or ending in *p_sweep*. Let *sit_last* be the last item in this sequence and let *sit_pred* and *sit_succ* be the items before and after the sequence, respectively; *sit_last* does not exist if there are no segments passing through or ending in *p_sweep*.

Sweeping through *p_sweep* reverses the subsequence starting with *sit_first* and ending with *sit_last* and hence two intersections can get lost, the intersection stored in *sit_last* and the intersection stored in *sit_pred*. The intersection stored in *sit_last* is with the segment associated with *sit_succ* and the intersection stored in *sit_pred* is with the segment associated with the successor of *sit_pred*. This is the item *sit_pred_succ*.

⟨*optimization, part 1*⟩≡

```
seq_item xit = Y_structure.inf(sit_last);
if (xit) { SEGMENT s1 = Y_structure.key(sit_last);
           SEGMENT s2 = Y_structure.key(sit_succ);
           inter_dic(s1,s2) = xit;
         }
```

⟨*optimization, part 2*⟩≡

```
seq_item xit = Y_structure.inf(sit_pred);
if ( xit )
{ SEGMENT s1 = Y_structure.key(sit_pred);
  SEGMENT s2 = Y_structure.key(sit_pred_succ);  // sit_first
  inter_dic(s1,s2) = xit;
  Y_structure.change_inf(sit_pred, seq_item(nil));
}
compute_intersection(X_structure, Y_structure,inter_dic,sit_pred);
sit = Y_structure.pred(sit_succ);
if ( sit != sit_pred )
  compute_intersection(X_structure, Y_structure,inter_dic,sit);
```

We also need to change the function *compute_intersection*. Before computing an intersection point we check whether the two segments already have an intersection event in the X-structure by a lookup in *inter_map*. If the lookup fails we compute the intersection and add it to the X-structure.

⟨*geometric primitives*⟩+≡

```
static void compute_intersection(sortseq<POINT,seq_item>& X_structure,
               sortseq&lt;SEGMENT,seq_item>& Y_structure,
               const map2&lt;SEGMENT,SEGMENT,seq_item>& inter_dic,
               seq_item sit0)
{ seq_item sit1 = Y_structure.succ(sit0);
  SEGMENT  s0   = Y_structure.key(sit0);
  SEGMENT  s1   = Y_structure.key(sit1);
  if ( orientation(s0,s1.target()) <= 0 &&
       orientation(s1,s0.target()) >= 0 )
  {
    seq_item it = inter_dic(s0,s1);
    if ( it == nil)
    { POINT q;
      s0.intersection_of_lines(s1,q);
      it = X_structure.insert(q,sit0);
    }
    Y_structure.change_inf(sit0, it);
  }
}
```

0.0.2 *Experimental Evaluation of the Sweep Line Algorithm*

We report about tests for *three kinds of test data*, namely random, difficult, and highly degenerate inputs, *three different implementations of points and segments*, namely the floating point kernel (FK), the rational kernel (RK) and the rational kernel with turned-off floating point filter (FK^{-}), and *with and without the optimization*. We describe the test data, list running times, and comment on the results.

Random Inputs: The random data set consists of n segments whose endpoints have random k bit coordinates. Table 0.1 gives the number of nodes and edges of the output graph and the running time for $n = 200$ and different values of k. The experiments indicate that the

k	V	E	RK$^-$	RK$^-$O	RK	RKO	FK	FKO
10	4813	9028	2.27	2	1.2	1.09	0.73	0.67
20	4742	8884	2.63	2.19	1.31	1.1	0.7	0.67
30	5467	10334	3.07	2.57	1.52	1.26	0.8	0.77
40	5478	10356	3.78	3.13	1.69	1.38	0.81	0.77
50	5168	9736	3.66	3.13	1.62	1.3	0.76	0.73
60	5558	10516	4.36	3.59	1.81	1.43	0.82	0.79
70	5909	11218	5.2	4.23	2.13	1.6	0.86	0.83
80	5174	9748	4.75	3.78	1.86	1.43	0.78	0.74
90	4808	9016	4.86	3.82	1.77	1.34	0.71	0.68
100	5080	9560	5.92	4.5	2.12	1.54	0.75	0.73

Table 0.1 200 random segments, coordinates are random k-bit integers. An "O" indicates the use of the optimization.

optimization described above and the floating point filter are effective. The optimization is more effective for the rational kernels because the computation of intersections is more costly in exact arithmetic. Floating point arithmetic is faster than exact arithmetic but the difference is never more than a factor of two in running time. We have to admit though that the difference can be made arbitrarily larger by choosing larger values of k.

Difficult Inputs: Let $size = 2^k$ and let $y = 2size/(n-1)$. The random data set consists of n segments where the i-th segment has endpoints $(size + rx1, size + i \cdot y + ry1)$ and $(3 \cdot size + rx2, 3 \cdot size - i \cdot y + ry2)$ and $rx1$, $rx2$, $ry1$, $ry2$ are random integers in $[-s, s]$ for some small integer s. For $s = 0$ all segments in the difficult data set pass through the point $(2 \cdot size, 2 \cdot size)$, and for small but non-zero values of s they intersect in the neighborhood of this point. Table 0.2 gives the results for the difficult data set with $s = 10$, $k = 10, 20, \ldots$, 100, and $n = 200$. The floating point filter and the optimization are again quite effective. The floating point implementation produced incorrect results for all values of k; the floating point implementation does, however, work correctly for smaller values of n and/or larger values of s.

Highly Degenerate Inputs: The highly degenerate test set consists of n segments with random coordinates in a small grid with side length s. For example, for $n = 100$ and $s = 10$ one should expect a large number of degeneracies. We used this test set to support our claim that the algorithm handles all degeneracies. We do not report running times for the highly degenerate inputs.

The readers may perform their own experiments by running either the sweep_time program in the demo directory or the sweep_segments_demo in xlman.

We were surprised by two outcomes of our experiments.

First, we expected the implementation using the rational kernel to be much slower than the floating point computation and not just by a factor of two. We achieve the small factor by

k	V	E	RK^-	RK^-O	RK	RKO	FK	FKO
10	20134	39669	9.84	8.29	5.07	4.46	error	error
20	20298	39997	11.75	9.71	5.65	4.64	error	error
30	20296	39994	12.33	10.5	6.04	4.88	error	error
40	20298	39997	14.79	11.71	6.5	5.13	error	error
50	20300	40000	16.12	12.5	6.7	5.22	error	error
60	20298	39997	16.32	12.95	6.91	5.45	error	error
70	20300	40000	18.77	14.84	7.51	5.69	error	error
80	20300	40000	19.82	15.91	7.62	5.72	error	error
90	20298	39997	21.27	16.25	7.68	5.71	error	error
100	20296	39994	24.61	18.39	8.58	6.24	error	error

Table 0.2 The difficult example with 200 segments. An "O" indicates the use of the optimization and error indicates that the computation with the floating point kernel gave the incorrect result.

the use of the floating point filter, by the optimization which avoids the costly recomputation of intersections, and by the observation that many equality tests for points can be replaced by tests for identity of points.

Second, we expected the floating point implementation to have difficulties with the difficult example. However, we were surprised by the fact that it never crashed. It always produced an output, albeit an incorrect one. We try to explain this phenomenon by arguing that the program does not crash as long as the sentinels are handled correctly, i.e, the segments *lower_sentinel* and *upper_sentinel* have all segments between them and all intersection points precede *pstop*. We do not care what the geometric tests do with segments that are not sentinels. If sentinels are handled correctly, every lookup in the Y-structure will return an item different from the first item in the Y-structure[7]. Also the walks performed in the Y-structure will determine a subsequence that does not include the sentinel items. For this reason none of the operations on the Y-structure will fail; i.e., it will never happen that we ask for the successor of the last or the predecessor of the first item. Also since *pstop* is handled correctly, we will never attempt to extract *next_seg* from an empty *seg_queue*.

[7] This sentence requires knowledge of the implementation of sorted sequences. The implementation is such that if the comparisons with the first and the last element of the sorted sequence are correct and the outcome of any other comparison is arbitrary, lookup will not return the first element.

Bibliography

[Meh84] K. Mehlhorn. *Data Structures and Algorithms 3: Multidimensional Seaching and Computational Geometry*. Springer Verlag, 1984.

[Mye85] E. Myers. An $O(E \log E + I)$ expected time algorithm for the planar segment intersection problem. *SICOMP*, pages 625 – 636, 1985.

Index